IN THE SPECIFICATION:

Each paragraph is listed first in its amended form per 37 CFR 1.121(b)(1)(ii).

1. Please replace the paragraph starting on page 1, line 24 with the following paragraph.

The idea of prefetching data is not new. Cahoon and McKinley (*see* B. Cahoon and K. S. McKinley, "Data Flow Analysis for Software Prefetching Linked Data Structures in Java," in Proceedings of 2001 International Conference on Parallel Architectures and Compilation Techniques, pp. 280-291, 2001) have researched extracting dataflow dependencies from Java applications for the purpose of prefetching state data associated with nodes. Wang et al. (*see* Z. Wang, T. W. O'Neil, and E. H-M. Sha, "Optimal Loop Scheduling for Hiding Memory Latency Based on Two-Level Partitioning and Prefetching," in IEEE Transactions on Signal Processing, pp. 2853-2864, 2001), when exploring how to best schedule loops expressed as dataflow graphs, also [[try]] tried to schedule the prefetching of data needed for loop iterations.

2. Please replace the paragraph starting on page 2, line 2 with the following paragraph.

Figure 2 illustrates a homogeneous and quasi-static Boolean Data Flow graph, according to Prior Art. While statically schedulable models of computation are common in signal and image processing applications and make efficient scheduling easier, the range of applications is restrictive because runtime scheduling is not allowed. Dynamically schedulable models of computation, such as Boolean dataflow, dynamic dataflow, and process networks, allow runtime decisions, but in the process make static prefetch difficult, if not impossible. As J. T. Buck explains in his thesis (*see* J. T. Buck, Scheduling dynamic dataflow graphs with Bounded Memory Using the Token Flow Model, PhD Thesis, U. C. Berkeley, 1993), Boolean Dataflow (BDF) is a model of

computation sometimes requiring dynamic scheduling. The *switch* and *select* actors allow conditional dataflow statements with semantics for control flow as shown in Figure 2.

3. Please replace the paragraph starting on page 2, line 13, with the following paragraph.

Cyclo-dynamic dataflow, or CDDF, as defined by Wauters et al. (*see* P. Wauters, M. Engels, R. Lauwereins, J.A. Peperstraete, "Cyclo-Dynamic Dataflow," p. 0319, 4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), 1996), extends cyclo-static dataflow to allow run-time, data-dependent decisions. Similar to a CSDF, each CDDF actor may execute as a sequence of phases *f1, f2, f3, f4, ..., fn*. During each phase *fn*, a specific code segment may be executed. In other words, the number of tokens produced and consumed by an actor can vary between firings as long as the variations can be expressed in a periodic pattern. CDDF allows run-time decisions for determining the code segment corresponding to a given phase, the token transfer for a given code segment, and the length of the phase firing sequence. The same arguments for using caller prediction for BDF graphs can be extended to include CDDF.

4. Please replace the paragraph starting on page 2, line 28, with the following paragraph.

Lee and Parks (*see* E. A. Lee and T. M. Parks, "Dataflow Process Networks," Proceedings of the IEEE, pp. 773-801, 1995) describe dataflow process networks as a special case of Kahn process networks. The dataflow process networks implement multiple concurrent processes by using unidirectional FIFO channels for inter-process communication, with non-blocking writes to each channel, and blocking reads from each channel. In the Kahn and MacQueen representation (*see* G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," Information Processing 77, Gilchrist

ed., North-Holland Publishing Co., 1977), run-time configuration of the network is allowed. While some scheduling can be done at compile time, for some applications most actor firings must be scheduled dynamically at run-time.

5. Please replace the paragraph starting on page 3, line 20, with the following paragraph.

Figure 4 illustrates two-level branch prediction. A '1' in the counter MSB predicts that a branch will be taken, while a '0' predicts that the branch will not be taken. This approach may achieve a prediction success rate in the 90% range. Patt and Yeh (*see* T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture, pp. 51-61, 1991) have tabulated the hardware costs to be significant for large history lengths.

6. Please replace the paragraph starting on page 10, line 20, with the following paragraph.

There are a few characteristics of applications that can take advantage of caller prediction model for prefetching node instance data. In one embodiment, applications should not execute in hard real-time since the prediction logic may be wrong, which may result in jitter. In one embodiment, applications for run-time caller prediction may include applications where quality of service (QoS) can vary depending on how fast the application is running. Buttazzo et al. (*see* G. Buttazzo and L. Abeni, "Adaptive Rate Control through Elastic Scheduling," in Proceedings of the 39th IEEE Conference on Decision and Control, pp. 4883-4888, 2000) point out that these applications are common due to the non-deterministic behavior of common low-level processor architecture components, such as caching, prefetching, and direct memory access (DMA) transfers. Buttazzo's work suggests voice sampling, image acquisition, sound generation, data compression, video playback, and certain feedback control systems as application

domains that can function at varying QoS depending on the execution rate of the application.

7. Please replace the paragraph starting on page 3, line 20, with the following paragraph.

Feedback control systems are particularly interesting because they may contain several subsystems that share common components. Abdelzaher et al. (*see* T. Abdelzaher, E. M. Atkins, and K. G. Shin, "QoS Negotiation in Real-Time Systems and Its Application to Automated Flight Control," in IEEE Transactions on Computers, pp. 1170-1183, 2000) explore a complex real-time automated flight control system that negotiates QoS depending on the load of the system. The flight control system consists of a main function that controls the altitude, speed, and bearing of the plane, and also contains subsystems for flight control performance. Continuous analysis of all of the inputs to achieve optimal control is a large computation task. However, the task of flying a plane can be accomplished with relatively little computation power by carefully reserving resources and by tolerating less than optimal control functionality. PID (Proportional-Integral-Derivative) control is a possible compelling application domain because PID tuning parameters are relatively insensitive to rate changes. Many measurement and control applications may display similar properties of varying acceptable QoS.

IN THE CLAIMS:

The following listing of claims will replace all prior versions, and listings, of claims in the application.

1. (Currently Amended) A method for run-time prediction of a next caller of a shared functional unit, wherein the shared functional unit is operable to be called by two or more callers out of a plurality of callers, the method comprising:

storing a caller history of the shared functional unit;

detecting a calling pattern of the plurality of callers of the shared functional unit, wherein said detecting comprises:

dividing the caller history into a first portion of the caller history and a second portion of the caller history, wherein the first portion and the second portion each hold a substantially equal amount ~~the same portion~~ of the caller history; and

comparing the callers ~~of~~ in the first portion of the caller history to the callers ~~of~~ in the second portion of the caller history;

predicting the next caller out of the plurality of callers of the shared functional unit; and

loading state information associated with the next caller out of the plurality of callers;

wherein the shared functional unit and the plurality of callers are operable to execute in parallel on a parallel execution unit.

2. (Original) The method of claim 1, wherein the run-time prediction is performed for an application described by a dataflow graph.

3. (Original) The method of claim 1, wherein the run-time prediction is performed for an application programmed in a dataflow language.

4. Cancelled.

5. (Currently Amended) The method of claim 1,

wherein said storing the caller history uses a history register, wherein the history register is operable to be divided into two substantially equal portions ~~parts~~.

6. (Currently Amended) The method of claim 5,

wherein said comparing operates to compare callers in the first portion ~~part~~ of the history register to the callers in the second portion ~~part~~ of the history register.

7. (Original) The method of claim 5,

wherein each of the plurality of callers has a unique identification, wherein the unique identification is operable to be used in the caller history.

8. (Currently Amended) The method of claim 7,

wherein the history register is operable to store the unique identification of each of the two or more callers calling the shared functional unit by operating analogously to a shift register.

9. (Original) The method of claim 7,

wherein said comparing the callers comprises comparing the unique identifications of the callers in the first portion of the caller history to the unique identifications of the callers in the second portion of the caller history.

10. (Currently Amended) The method of claim 1,

wherein said comparing the callers in the first portion ~~part~~ of the caller history to the callers in the second portion ~~part~~ of the caller history operates to select a periodic portion of the caller history.

11. (Currently Amended) The method of claim 10, further comprising:

using a multiplexer to said predict the next caller of the shared functional unit after selecting the periodic portion of the caller history.

12. (Previously Presented) The method of claim 1,

wherein the parallel execution unit comprises one or more of:

an FPGA;

a programmable hardware element;

a reconfigurable logic unit;

a nonconfigurable hardware element;

an ASIC;

a computer comprising a plurality of processors; or

any other computing device capable of executing multiple threads in parallel.

13. (Previously Presented) The method of claim 1,

wherein the state information comprises one or more of:

execution state;

values of any variable;

previous inputs;

previous outputs; or

any other information related to execution of a node in a dataflow diagram.

14. (Currently Amended) The method of claim 1,

wherein a dataflow program comprises a plurality of nodes, wherein the plurality of nodes comprises the shared functional unit, wherein the plurality of nodes further comprises the plurality of callers,

wherein the run-time prediction operates to optimize execution of the plurality of nodes in the dataflow program.

15. (Currently Amended) The method of claim 14,

wherein the shared functional unit and the plurality of callers are generated using the ~~from a~~ dataflow program.

16. (Currently Amended) A method for run-time call prediction for resolving resource contention between two or more callers of a shared node in a dataflow program, the method comprising:

storing a caller history of the shared node;

detecting a calling pattern by a plurality of callers of the shared node ~~functional unit~~, wherein said detecting comprises:

dividing the caller history into a first portion of the caller history and a second portion of the caller history, wherein the first portion and the second portion each hold a substantially equal amount ~~the same portion~~ of the caller history; and

comparing the callers ~~of~~ in the first portion of the caller history to the callers ~~of~~ in the second portion of the caller history;

predicting a next caller out of the plurality of callers of the shared node ~~functional unit~~; and

loading state information associated with the next caller out of the plurality of callers;

wherein the shared node ~~functional unit~~ and the plurality of callers are operable to execute in parallel on a parallel execution unit.

17. (Currently Amended) The method of claim 16,

wherein the dataflow program comprises ~~of~~ a plurality of nodes, wherein the plurality of nodes comprises the shared node, wherein the plurality of nodes further comprises the plurality of callers ~~wherein one or more of the plurality of nodes are operable to be called by two or more nodes of the plurality of nodes~~.

18. (Currently Amended) The method of claim 17 ~~16~~,

wherein the run-time call prediction operates to optimize execution of the plurality of nodes in the dataflow program.

19. (Currently Amended) The method of claim 16,

wherein the dataflow program executes on the ~~a~~ parallel execution unit, wherein the parallel execution unit comprises one or more of:

an FPGA;

a programmable hardware element;

a reconfigurable logic unit;

a nonconfigurable hardware element;

an ASIC;

a computer comprising a plurality of processors; or

any other computing device capable of executing multiple threads in parallel.

20. Cancelled.

21. (Previously Presented) The method of claim 16,

wherein each of the plurality of callers has a unique identification, wherein the unique identification is operable to be used in the caller history.

22. (Currently Amended) The method of claim 21,

wherein the caller history ~~register~~ is operable to store the unique identification of each of the two or more callers calling the shared node ~~functional unit~~ by operating analogously to a shift register.

23. (Original) The method of claim 21,

wherein said comparing the callers comprises comparing the unique identifications of the callers in the first portion of the caller history to the unique identifications of the callers in the second portion of the caller history.

24. (Currently Amended) The method of claim 16,

wherein said comparing the callers in the first portion ~~part~~ of the caller history to the callers in the second portion ~~part~~ of the caller history operates to select a periodic portion of the caller history.

25. (Currently Amended) The method of claim 16,

wherein the shared node ~~functional unit~~ and the plurality of callers are generated by ~~from~~ the dataflow program.

26. (Currently Amended) A memory medium comprising instructions to generate a program to perform run-time call prediction of a next caller of a shared functional unit, wherein the program is intended for deployment on a parallel execution unit, wherein the program is executable to:

store a caller history of the shared functional unit ~~node~~;

detect a calling pattern of a plurality of callers of the shared functional unit, wherein the shared functional unit is operable to be called by two or more callers out of the plurality of callers, wherein said detecting comprises:

dividing the caller history into a first portion of the caller history and a second portion of the caller history, wherein the first portion and the second portion are each operable to hold ~~substantially the same portion~~ an equal amount of the caller history; and

comparing the callers ~~of~~ in the first portion of the caller history to the callers ~~of~~ in the second portion of the caller history;

predict the next caller out of the plurality of callers of the shared functional unit; and

load state information associated with the next caller out of the plurality of callers;

wherein the shared functional unit and the plurality of callers are operable to execute in parallel on the parallel execution unit.

27. (Currently Amended) The memory medium of claim 26,

wherein a dataflow program comprises a plurality of nodes, wherein the plurality of nodes comprises the shared functional unit, wherein the plurality of nodes further comprises the plurality of callers,

wherein the run-time call prediction operates to optimize execution of the plurality of nodes in the dataflow program.

28. (Currently Amended) The memory medium of claim 26,

wherein the program executes on a the parallel execution unit, wherein the parallel execution unit comprises one or more of:

an FPGA;

a programmable hardware element;

a reconfigurable logic unit;

a nonconfigurable hardware element;

an ASIC;

a computer comprising a plurality of processors; or

any other computing device capable of executing multiple threads in parallel.

29. Cancelled.

30. (Currently Amended) The memory medium of claim 26,

wherein said storing the caller history comprises storing the caller history in a history register;

wherein said comparing operates to compare callers in the first portion of the history register to the callers in the second portion of the history register;

wherein each of the plurality of callers has a unique identification~~, wherein the unique identification is operable to be used in the caller history~~.

31. (Currently Amended) The memory medium of claim 30,

wherein the history register is operable to store the unique identification of each of the two or more callers calling the shared functional unit by operating analogously to a shift register.

32. (Original) The memory medium of claim 30,

wherein said comparing the callers comprises comparing the unique identifications of the callers in the first portion of the caller history to the unique identifications of the callers in the second portion of the caller history.

33. (Currently Amended) The memory medium of claim 26,

wherein said comparing the callers in the first ~~part~~ portion of the caller history to the callers in the second ~~part~~ portion of the caller history operates to select a periodic portion of the caller history.

34. (Previously Presented) The memory medium of claim 26,

wherein the program comprises one or more of:

program instructions;

digital logic; or

any type of hardware description used to configure the parallel execution unit.

35. (Currently Amended) The memory medium of claim 26,

wherein the shared functional unit and the plurality of callers are generated ~~from~~ by the program.

36. (Original) The memory medium of claim 26,

wherein the program comprises a control and arbitration logic unit that is operable to said detect, said predict, and said load.

37. (Currently Amended) A system for run-time optimization of a dataflow program, the system comprising:

a parallel execution unit;

a plurality of callers;

a shared functional unit, wherein the shared functional unit is operable to be called by two or more callers out of the plurality of callers, wherein the shared functional unit and the plurality of callers are operable to execute in parallel on the parallel execution unit;

an optimization algorithm, wherein the optimization algorithm is operable to:

store ~~storing~~ a caller history of the shared ~~node~~ functional unit;

detect a calling pattern of the plurality of callers of the shared functional unit, wherein said detecting comprises:

dividing the caller history into a first portion of the caller history and a second portion of the caller history, wherein the first portion and the second portion are each operable to hold ~~substantially the same portion~~ an equal amount of the caller history; and

comparing the callers ~~of~~ in the first portion of the caller history to the callers ~~of~~ in the second portion of the caller history;

predict ~~the~~ a next caller out of the plurality of callers of the shared functional unit; and

allocate state information associated with the next caller out of the plurality of callers.

38. (Previously Presented) The system of claim 37,

wherein the parallel execution unit comprises one or more of:

an FPGA;

a programmable hardware element;

a reconfigurable logic unit;

a nonconfigurable hardware element;

an ASIC;

a computer comprising a plurality of processors; or

any other computing device capable of executing multiple threads in parallel.

39. Cancelled.

40. (Currently Amended) The system of claim 37,

wherein said storing the caller history comprises storing the caller history in a history register;

wherein said comparing operates to compare callers in the first portion of the history register to the callers in the second portion of the history register;

wherein each of the plurality of callers has a unique identification~~, wherein the unique identification is operable to be used in the caller history~~.

41. (Currently Amended) The system of claim 40 ~~37~~,

wherein the history register is operable to store the unique identification of each of the two or more callers calling the shared functional unit by operating analogously to a shift register.

42. (Currently Amended) The system of claim 40 ~~37~~,

wherein said comparing the callers comprises comparing the unique identifications of the callers in the first portion of the caller history to the unique identifications of the callers in the second portion of the caller history.

43. (Currently Amended) The system of claim 37,

wherein said comparing the callers in the first ~~part~~ portion of the caller history to the callers in the second ~~part~~ portion of the caller history operates to select a periodic portion of the caller history.

44. (Currently Amended) The system of claim 37,

wherein the shared functional unit and the plurality of callers are generated ~~from~~ using the dataflow program.

45. (Currently Amended) The system of claim 37,

wherein the optimization algorithm is implemented ~~comprised~~ on a control and arbitration logic unit that is operable to said detect, said predict, and said allocate ~~load~~.